# METHOD AND SYSTEM FOR AUTOMATICALLY GENERATING A GLOBAL SIMULATION MODEL OF AN ARCHITECTURE

[0001] The invention concerns a method and a system for generating a global simulation model of an architecture. More particularly, the invention concerns a configuration method and a system called a "Configurator" for implementing the method.

[0002] With the increasing complexity of hardware systems, it is necessary, in order to verify systems or integrated circuits under design, to be able to handle configurations that increasingly include models written in a hardware description language, for example of the HDL type (such as VDHL or Verilog, the most widely used), and in a high level languages of the HLL type (such as C or $C^{++}$); these languages describe, on the one hand, the elements constituting the hardware, and on the other hand, the models constituting the simulation environment.

[0003] The term "Configuration" will be used to refer to a set of software models of elements called "Components," constituting a global simulation model.

[0004] The invention can be used to verify the design of ASICS by simulating their operation, for example in an environment identical or very similar to their end use, the configuration method making it possible to choose components and their software models from a plurality of available models in order to create simulation configurations. In the prior art, the configurations are rigid, and traditionally prepared "by hand" using text editors or graphics editors, based on a predefined list of possible configurations. Each modification in the model in HDL language requires manual corrections to be incorporated into all of the configurations. This happens several times during the development of an ASIC and constitutes a source of errors and modification problems, resulting in production delays. Modifications of a configuration are often sources of errors that are hard to find, as the size of some configurations can reach tens of thousands of lines, making them difficult to handle manually.

[0005] The time required to write and debug a configuration can be very long, making it difficult to generate and add new configurations into the environment. For that reason, when an environment contains a lot of elements for which it is difficult to predict all of the usable configurations, the use of certain configuration variants that facilitate debugging (simplified target configurations, for example) is often avoided.

[0006] These problems are accentuated in the more and more widely used co-simulation environments, wherein the models come from different sources and are written in high level programming languages (HLL) such as C or $C^{++}$, or in hardware description languages (HDL) such as VERILOG or VHDL. For the same component of the configuration, there are often several models (ex: functional, in high level programming languages; behavioral, in HDL, synthesizable HDL, etc.) that it would be desirable to be able to use transparently, as needed.

[0007] Moreover, the models to be connected often have, at the level of the connecting interfaces, differences requiring the use of adaptation modules. This is the case, for example, for circuits with sophisticated input/output interfaces for which the logical level of the protocol is simulated first, the physical level being developed near the end of the project. The ability to choose adaptation models for the variants of HDL interfaces corresponding to the various development phases of the project constitutes an additional degree of freedom that complicates the writing of the configurations even more. Another problem stems from the fact that mixed HDL (for example Verilog or VHDL)/HLL (for example C++) type models developed separately must be updated coherently. In the case of nonautomated management, this is a potential source of errors.

[0008] The object of the present invention is to limit one or more of these drawbacks.

[0009] To this end, the invention primarily concerns a method of automatic generation, by means of a data processing system associated with a program called a Configurator for creating a global simulation model of an architecture comprising models of integrated circuits under development that can constitute, with the help of the automatic Configurator, a machine or a part of a machine, and environment simulation models that make it possible to test and verify the circuit under development, a configuration definition file for components of the architecture, these components constituting fixed functional blocks for describing the functionalities of integrated circuits or parts of integrated circuits, the components being chosen by the user from a library of various component types and a library of environment components, in order to create the global model of the architecture corresponding to the functional specification defined in the configuration definition file and conforming to the specification of the architecture of the global model specified by an architecture description file, a method characterized in that it includes the following steps :

- the reading of the architecture description file of the global model and the storage, in a component and connection rule table, in a connection coherency rule table, and in a source file formatting table, of information related to all of the possible configurations, each component obtaining an name that unambiguously identifies its position in the architecture, and a type from among several types (Active Components, Monitoring and Verification Blocks, Intermediate Blocks, System Blocks and Global Blocks),

- the instantiation of the components specified in the configuration definition file by the user-developer using a list of the components present, designated by their names and types and including parameters or invoking procedures, the configuration definition file comprising a file from which to select the components and their types and optional additional indications concerning the type of interface and the server involved in the configuration to be generated by the Configurator, and the storage of the corresponding information in an instance connection table,

- the topological connection of the instances and the storage of the corresponding information in an instance connection table,

- the physical connection of the interface signals, at the level of each instance of the components, by applying regular expressions, stored in the component and connection rule table, based on the names of the signals constituting a wiring table,

- the use of the instance connection table, the wiring table and the formatting table to automatically generate HDL-type and HLL-type source files of the global simulation model corresponding to the configuration specified by the configuration definition file.

[0010] According to another characteristic of the method, the Configurator system transmits to the HLL-type parts of each component information on:

- the name of the component;
- the type of the instance
- the HDL path, i.e. the hierarchical name of the component in the description of the model.

[0011] According to another characteristic of the method, the configuration definition file also includes a keyword indicating the name or number of the server in which a component is instantiated, when the method is used in a multi-server system.

[0012] According to another characteristic of the method, in the case of a multi-server utilization, the Configurator system executes the following steps:

- the division of the Configuration into several (HDL-type and HLL-type) parts, sorting the HDL-type components and the HLL-type objects according to the servers to which they belong,
- the generation of the HDL-type peripheral components used for sending and receiving signals between the parts of the configuration,
- the duplication of the Global Blocks by the Configurator system and the instantiation of the Global Blocks duplicated in each server,
- the generation of HLL-type parts that serve as a communication medium between the servers.

[0013] According to another characteristic of the method, the automatic connection between the components by the Configurator system includes several phases:

- a high-level topological phase for selecting the components and their respective positions,
- a wiring phase for creating the actual connection between the components, this phase generating as a result a wiring table that associates the signals connected to one other with the unique name of the wire that connects them,
- a phase for generating HDL-type and HLL-type source files.

[0014] According to another characteristic of the method, the wiring phase is performed by the Configurator system in the following three steps:

a. the Global Blocks and the System Blocks are first connected to all of the components,
b. next come the connections of the signals between the other components,
c. after the wiring, an additional pass makes it possible to connect the remaining unconnected signals of each component to predetermined signals in order to produce a given stable state; the Configurator system then generates partial configurations comprising a subset of the architecture.

[0015] According to another characteristic of the method, the predetermined signals are the signals of the System Block corresponding to the component.

[0016] According to another characteristic, the description file of the architecture of the global model includes the simulation models of the.

Global Blocks and the System Blocks, these two types of components being connected to one another and handling environment signals.

**[0017]** According to another characteristic, the System Blocks are connected to the other components and supply them with the system signals that are specific to them.

**[0018]** According to another characteristic of the method, the data processing system performs a conformity check of the connections, comparing the connection table of the real instances between blocks to the connection coherency rule table.

**[0019]** According to another characteristic of the method, the data processing system compares the physical connections between the components to the connection coherency rule table, in order to detect any incompatibilities between the ends of the connections between the components, and in such a case, it specifies, and adds into the instance connection table, an adapter component inserted into the connection in question.

**[0020]** According to another characteristic, the configuration definition file includes information, specified by an attribute, concerning the utilization of adapter components with the instances of the active Components, whose connections are compared to the instance connection table in order to detect any incompatibilities between the ends of the connections between the components, and in such a case it specifies, and adds into the instance connection table, another adapter component inserted into the connection in question. According to another characteristic of the method, the data processing system selects certain connections between the components of the connection coherency rule table and specifies, and adds into the instance connection table, additional connections constituting branches leading to respective additional models, which represent tools for monitoring the connections.

**[0021]** According to another characteristic of the method, in the source file generation phase, the Configurator system generates the source files in HDL language and in HLL language, based on the content of the component and connection rule table, the connection coherency rule table, the source file formatting table, the instance connection table and the wiring table.

**[0022]** According to another characteristic of the method, the data processing system executes an operation through the Configurator system for each configuration variant, in order to obtain several simulation models corresponding to the same functional specification, but written in a description comprising various mixtures of languages of different levels (HDL, HLL).

[0023] According to another characteristic of the method, the data processing system generates the functional specification of the global simulation model in a computer format compatible with a high-level programming language, (HLL) and in a format compatible with a hardware description language (HDL).

[0024] According to another characteristic of the method, the configuration definition file comprises, for each component, at least one part in HDL-type language, said part in HDL-type language providing an interface with other models.

[0025] According to another characteristic of the method, the models that include a part in HLL-type language include interface adapters.

[0026] According to another characteristic, the Configurator system chooses each interface adapter model as a function of the connection coherency rule table.

[0027] According to another characteristic of the method, the connections of the physical signals are specified by "Ports," each port being an arbitrary selection of signals from the HDL-type interface of a component by means of regular expressions based on the names of these signals, and being constituted by regular expression/substitute expression pairs, these expressions being successively applied to the name of each signal of the HDL-type interface, and if the final substitution is identical for two signals, the latter are connected to one another, the connection being stored in the wiring table.

[0028] According to another characteristic of the method, each interface adapter being shared among several models connected to the same port, only one of these models transmits signals through said port.

[0029] Another object of the invention is to offer a Configurator system that makes it possible to eliminate one or more of the preceding drawbacks.

[0030] This object is achieved by the data processing system for automatically generating a global simulation model of a configuration of fixed functional blocks, mutually connected by interworking connections so as to constitute the global simulation model of an architecture comprising models of integrated circuits under development that can constitute a machine that conforms to the functional specification of a configuration, this system being characterized in that the data processing system uses a

Configurator program that includes means for creating a simulation of the wiring by applying stored regular expressions, and for using a configuration definition file in a high level language, a component and connection rule table describing the properties (type, HDL-type interfaces, ports, constructors of HLL class objects, etc.) of the software components for simulating the circuit, a connection coherency rule table in a high level language (HLL), means for instantiating the elements resulting from the configuration definition file, and an HLL code generator that combines the parameters of the components with the connection rules. According to another characteristic of the system, there are at least five types of components: Active Components, Monitoring and Verification Blocks, Intermediate Blocks, System Blocks and Global Blocks.

**[0031]** According to another characteristic of the system, it is equipped to perform a conformity check of the connections by comparing the instance connection table with a table of coherency rules for the physical connections between the models chosen from the blocks constituting the global model.

**[0032]** According to another characteristic of the system, it is equipped to compare the connection table of the real instances between blocks to the connection coherency rule table, in order to detect any incompatibilities between the ends of the connections between blocks, and in such a case to specify, and add into the coherency rule table of the real connections, a functional adapter block inserted into the connection in question.

**[0033]** According to another characteristic of the system, the component and connection rule table, which includes the properties of the components, contains global parameters common to all of the component types and exists in the form of a table distributed into one or more tables, which may or may not be associative, wherein the entries are names designating all of the possible models for the same component.

**[0034]** According to another characteristic of the system, the associative tables can contain the description, either in the form of parameter sets or in the form of references to procedures that generate the required values, the entries of these associative tables being names designating all of the possible models for the same component and forming a character string containing predetermined special identifiers, replaced by the values calculated by the Configurator system.

[0035] According to another characteristic of the system, at least three selectors indicate the instance to be used, the latter being transmitted as a parameter to a constructor of an HLL object.

- a first selector indicates the current instance,
- a second selector specifies the instance connected to the other end of the port,
- a third selector indicates the composite instance corresponding to the active Component containing the observation port.

[0036] According to another characteristic of the system, the Configurator system uses one or more connection coherency rule tables, which represent the rules for interconnecting the components and for inserting intermediate components, one or more component and connection rule tables, which represent the system-level connection rules and the rules for generating connections between the signals, and one or more source file formatting tables, which represent the rules for generating instances of HLL-type objects.

[0037] According to another characteristic of the system, the Configurator system uses:

- an HLL base class for uniquely identifying each object instantiated and for polling the configuration,

- means for generating and automatically instantiating System Blocks,

- means for tables associating the signals connected together under the unique name of the connecting wire,

- means for using a formatting table to generate the source files in HDL and HLL.

[0038] According to another characteristic of the system, the operator functionally specifies the configuration in the highest level language as much as possible, and completes the functional specification with the components in the lowest level language.

[0039] According to another characteristic of the system, the following entries in the hash define the component Type (for example DUT (HDL model), XACTOR (transactor), MONITOR, VERIFIER or other types), and corresponds each Component Type to a hash, in turn composed of the following entries:

– a first entry *ReqModule* contains the name of the HDL module of the component and the name of the corresponding source file,

– a second entry *Connect* is the definition of the method for selecting the signals that are part of a Port, this description being composed of a set of entries indexed by the name of the Port; the configurator associates each Port name with a table of regular expressions and a pointer to a signal connection procedure that controls the application of these expressions to the names of the signals of the interface of the component.

[0040] According to another characteristic of the system, the generic structure of the active Components includes a Block containing the HDL description and a Block in HLL that provides the access paths to the HDL resources, and if necessary, a description of the block in HLL; the set of signals of the HDL block constitute the interface of the containing Block, formed by Ports, which are arbitrary logical selections of the signals of an interface, and by interface adapters, which are the software parts that handle, in each Port, the two-way communication between the parts in HLL and those in HDL, the interface adapters being chosen by the Configurator system.

[0041] According to another characteristic of the system, the Ports are specified in the form of regular expressions, which serve both to select the subsets of signals to be connected and to define the connection rules.

[0042] According to another characteristic of the system, the Configurator system generates so-called Transfer Components, which are inserted on each side of the cutoff to the servers, these components simply being wires for the inputs and registers for the outputs.

[0043] The elementary model components, which are shared among the Composite Model Components or the Global Blocks belonging to the various servers, are automatically duplicated by the Configurator system and instantiated in each server.

[0044] The invention will be better understood with the help of the following description of a preferred embodiment of the method of the invention, in reference to the attached drawings, in which:

- Fig. 1 represents, in highly schematic form, an exemplary architecture of the global simulation model for an integrated circuit (ASIC) under development, called BRIDGE (12);
- Figs. 2A through 2D are diagrams illustrating the various components of the Configurator system and the steps for implementing these components in the method of the invention;

- Figs. 3a through 3c represent various stages in the modeling of a circuit using a mixed HDL (VERILOG or VDHL) type and the HLL ($C^{++}$) type mode;.
- Fig. 4 represents the generic structure of an elementary Model;
- Fig. 5 represents the generic structure of a Composite Model;
- Fig. 6 describes the correspondence between the tables of the description of the configurator system in Figs. 2a through 2d and the tables for implementing the method of the invention;
- Figs. 7a through 7k schematically represent the various models of the components with the necessary variants of their interfaces and description levels for the configurations related to the architecture in the example of Fig. 1;Figs. 8a through 8e represent the successive configurations of the architecture that results in the creation of a given model, the final model of which corresponds to that of Fig. 8e, for which all of the HDL-type models of the circuit under design are available, this final model corresponding to the configuration; Fig. 8f represents the distribution of the simulation configuration of the model of Fig. 8a, for example into two machines.

[0045] The invention concerns a system called a "Configurator," and the configuration method implemented by the system.

[0046] A global simulation model is typically composed of one or more models of integrated circuits under test (DUTs) surrounded by models that create a testing and verification environment. These models create complex stimuli and receive complex responses from the model being tested. These components can be transactors (XACTORS) – models that generally have a program interface (API) that permits control by tests outside the model, these tests generally being written in a high level language (HLL).

[0047] The verification environment can also contain components called Monitoring Blocks (MONITOR) and components called Verification Blocks (VERIFIER). These components are not directly involved in the exchange of signals between the other components of the global simulation model, but are used to observe and interpret them. The Monitoring Blocks (MONITOR) serve as analysis aids for the tests; they have program interfaces (APIs) for reporting events observed in the signals of the global model. The Verification Blocks (VERIFIER) are components that have a reference specification for the operation of the model being tested, and by observing the signals of the global simulation model, they are capable of verifying the proper operation of the model.

[0048] Fig. 1 represents an exemplary architecture of an integrated circuit system under development for which it is necessary, in a first step, to debug the global simulation model corresponding to Fig. 8c, chosen in order to illustrate the greatest number of mechanisms implemented by the configurator. It should be clear that the steps in Figs. 8a, 8b, could be executed first, depending on the availability of the models and the objectives of the verification. The final model desired is the one that corresponds to Fig. 8e. The architecture is constituted by a processor (CPU) communicating through a bridge (BRIDGE) with a system memory (MEMORY) and input/outputs (I/O). The model produced by the "Configurator" system of the invention is constituted by a processor component CPU of the XACTOR type, connected by an interface of the "fbus_p" type to an intermediate block (fbus_xchg) 101 having a different type of interface. Another intermediate block (fbus_xchg) (102), in Figs. 8b and 8c, connects the first intermediate block (101) to a bridge-type component (BRIDGE) of the DUT_CORE type that communicates with a model, written primarily in an HDL-type language, of a memory (MEMORY) of the DUT type, with a model written primarily in an HDL-type language of an input/output (I/O) of the DUT type, and with a system block (SYS_BRIDGE).

[0049] The "Configurator" system of the invention handles the connection of software simulation elements called components for the purpose of simulating hardware circuits.

[0050] There are at least 5 classes of components:

- the "Active Components" (see below);
- the "Monitoring and Verification Blocks" (see below);
- the "Intermediate Blocks" (see below);
-the "System Blocks" (see below);
- the "Global Blocks: (see below and 1 of A2).

[0051] Each type of component can have several variants of embodiment of the same element, differentiated by the description level (see below) or by the signals in the interfaces (see below). Each type of Component can be described in several levels of detail (functional, behavioral, gates, etc.), in an HDL-type language like VERILOG or VHDL, or in a high level language (HLL) like C or $C^{++}$, complemented by an HDL-type interface.

**[0052]** Several description levels for describing similar functionalities can coexist and can have HDL-type interfaces that are similar but not necessarily identical. Certain descriptions can have more functionalities, and the HDL-type interfaces can contain completely different sets of signals.

**[0053]** The components are connected based on a predetermined schema that is considered to be fixed for each execution of the Configurator system. These connections are defined by the architecture of the global simulation model and specified by the architecture description file (FDARCH) (see Annexes A1-A5).

**[0054]** Each instance of a component in this schema obtains a name or label that unambiguously identifies the position of the component and its type.

**[0055]** The definition file of the configuration (FCONF) provides a synthetic description of the Configuration variant to be generated by the Configurator system. Only the main components (Active Components, Monitoring Blocks and Verification Blocks) are mentioned in it, along with the types of models desired. The other components (Global Blocks, System Blocks and Intermediate Blocks) are instantiated automatically by the Configurator system.

**[0056]** Among the various types of components, the "Active Components" constitute the subset of the objects explicitly designated in the configuration file (FCONF) that exchange stimuli with their environment by transmitting and receiving.

**[0057]** Among the various types of components, the "Monitoring and Verification Blocks" constitute the subset of the objects explicitly designated in the configuration file (FCONF) that merely receive information from the environment. They are used for purposes of observation and Verification (MONITOR, VERIFIER). The operation granularity of these models is the event, which reports changes in control signal values and arbitrary data exchanges.

**[0058]** All the other Components constitute so-called implicit components, which are managed and instantiated automatically by the Configurator system, as a function of the parameters of the configuration file (FCONF) (explicit component type, interface type, etc.).

[0059] The components can be connected to each other directly or via external adaptation components called "Intermediate Blocks," specially defined and declared for this purpose. They are often used, as will be seen below, during successive development phases to complete the missing parts of the design.

[0060] The Configurator system may thus insert one or more intermediate blocks to connect two active components.

[0061] The components called "System Blocks" are associated with the other components in order to supply them with the environment signals that are specific to them. These components encapsulate, at the level of each interface, all of the system signals that do not participate in the connection between the other components. The "System Blocks" themselves are connected to the "Global Blocks," and manage all of the environment signals, i.e., the clock signals and general control signals (clock, reset, powergood, diagnostic), which may be transformed in order to adapt them to the needs of the corresponding active block (polarity change, delay, etc.), as well as the specific signals that are different for each particular instance of the active component in question, for example the signals that encode the module number or the operating mode of the component, etc. The latter are managed by parameters provided by the Configurator system to the instances of the models of the components generated. If, for a given Component, the System Block is not necessary (which is indicated by the description tables of the configuration), it will be connected directly to the Global Blocks (see 1 of A2).

[0062] The configuration file (FCONF) can contain additional information specifying intermediate blocks to be used in association with the instances of the active components. Thus, the user can influence the choice of the intermediate component, or eliminate the ambiguity if there are several possible choices. The connections thus obtained are compared to the connection coherency rule table (TRCOH) in order to detect any incompatibilities between the ends of the connections between the components, and in such a case to choose, and add into the instance connection table (TCINST), another adapter component (Intermediate Block), inserted into the connection in question.

[0063] The Configurator system is based on a generic representation, as described in Figs 3a through 3c, common to all of the components declared in the architecture description file (FDARCH) of the global simulation model and comprising two parts, of the HDL type (for example VERILOG or VHDL) and HLL type (for example $C^{++}$).

[0064] In the case of a component described entirely in an HDL-type language (Fig. 3a), the HLL-type part is reduced to one instance, which makes it possible to indicate its presence in the Configuration during the simulation and supplies the paths for access to the HDL-type resources of the component.

[0065] For components described in an HLL-type language (Fig. 3c), it is the HDL-type part that is reduced to a strict minimum and is limited to the description of the interface registers and signals.

[0066] All of the intermediate levels between these two extremes are possible, and are naturally used in the context of processes for developing ASIC circuits.

[0067] Typically, during development, one begins with an HLL-type model with a minimal HDL-type interface, then moves on to increasingly complete HDL-type models written by the designers, and ends up with models on the logic gate level that are automatically synthesized from HDL-type models.

[0068] Mixed models are often used because they allow for a precise and efficient simulation by dedicating the high-level language (HLL) to complex modelings for which the HLL-type language offers a compact and quickly executed solution. Nevertheless, even for models written primarily in an HLL-type language, the HDL-type part can be non-trivial because of performance losses due to changes in the simulation context between the HDL-type and HLL-type parts, respectively. A typical example is an interface whose signals change, even without any real data transfer activity. In this case, it is preferable to process the signals in the HDL-type part of the model, and to switch to the HLL-type part when the data are present in the interface.

[0069] The interface of a component is the set of all the signals present on its periphery. For components described in an HDL-type language only, this corresponds to the external interface of the definition of this component in an HDL-type language (for example VERILOG or VHDL). For components defined in a mix of HLL- and HDL-type languages, the interface of a component is the sum of the signals of all the HDL-type models used on the periphery of this component.

[0070] Fig. 4 describes the generic structure of the elementary models that constitute the majority of the components. This structure typically, though not exclusively, applies, in the case of the ASIC circuit under development, to interface adapters, intermediate blocks, and system blocks.

[0071] It includes a containing block, marked C, containing the HDL-type description marked A, and the HLL Block marked B that provide the paths for access to the HDL-type resources and, if necessary, a description of the block in an HLL-type language. The set of signals of the HDL-type block constitutes the interface of the block C. For purposes of connecting between the blocks, we will use the concept of Ports (see below), which are arbitrary logical selections of the signals of an interface. It is possible for a signal to belong to several Ports at once.

[0072] The Configurator system is capable of handling so-called "composite" models comprising a part in an HLL-type language and including other models of components called "interface adapters." Interface adapters are mixed HDL/HLL-type models having a program interface (API) in an HLL-type language through which they communicate with the composite model. Composite models are particularly useful for simulation environment components (for example MONITOR, VERIFIER, XACTORS) wherein the model must adapt to signals present in the different variants of the models used in a configuration.

[0073] Fig. 5 describes the generic structure of composite models, used especially for modeling the components of the simulation environment. Seen from outside, the composite model is identical to the elementary model. Inside, the HLL specification of the model (marked D) communicates with the external HDL-type interface through interface adapters (marked $C\_Port_i$), modeled by means of elementary structures in an HLL-type language (marked B_PORT), and in an HDL-type language (marked A_PORT).

[0074] The Configurator system is equipped to automatically select the interface adapters for a composite model. The Configurator system examines the list of usable interface adapters for a given composite model described by the properties of the model, and chooses the interface adapter model whose physical connections with the rest of the global model conform to the connection coherency rule table (TRCOH). This approach makes it possible to quickly adapt to new types of interfaces by adding new adapters.

[0075] It is important to emphasize that the same component can be modeled by either an elementary or a composite structure, depending on its description level.

[0076] An interface adapter component can be shared among several composite models connected to the same port. Only one of these models is authorized to transmit signals to the port, the other models being purely observers. A typical utilization involves the Monitoring and Verification Blocks, which do not send output signals. The sharing of the interface adapters speeds up the simulation by simplifying and factoring parts of the model.

[0077] Hereinafter, the interface adapters that observe the signals on behalf of the Monitoring or Verification Blocks that are Composite Models will be called Probes.

[0078] The Configurator system is designed to optimize the utilization of the Probes while keeping their numbers to a minimum. Since the description of the Components establishes the notion of equivalency between certain Components, the Configurator system first tries to share the port of an "Active Component." If this proves impossible, it instantiates a suitable Probe component that can be connected to the HDL-type interface from a list provided in the description of the Monitoring or Verification Component.

[0079] The section below explains the definitions related to the connections between Components. The concepts of interfaces and ports are used to support the description of the connections between the components.

[0080] Let us recall that the port is an arbitrary selection of signals of the HDL-type interface of a component, performed using regular expressions based on the names of these signals. A given signal can belong to one or more ports. The definition of each port is constituted by pairs of regular expressions and corresponding substitute expressions. These expressions are successively applied to the name of each signal of the interface, and in case of an appropriate "match," the substitute expression is applied, and the name thus obtained moves on to the next substitution operation. If the final substitution gives a final result that is identical for two signals, the latter will be connected to one another. The configurator generates a unique name for the connection and places the appropriate information in the wiring table (TCAB). The method described makes it possible to express complex connection rules between two components. For example, it is possible to connect signals with different names, or to create rules for connecting the input signals with the output signals.

[0081] The creation of this method for definition by the interfaces and the ports makes it possible to separate the declarations of HDL-type models and the specifications of connections for the Configurator system. The latter combines these two sources of information in order to generate the connections. In the majority of cases, modifying declarations in the HDL-type parts, which happens quite frequently during development, does not entail any modifications in the regular expressions that describe the wiring.

[0082] In the exemplary embodiment of the invention discussed below, only the point-to-point connections are described. The "bus"-type connections are easily modeled using a component with several outputs incorporating the bus.

[0083] The method of automatic connection between the "components" by the Configurator system in order to create the global simulation model will be described below. This method comprises the following steps:

- The reading of the architecture description file (FDARCH) of the global model and the storage, in a component and connection rule table, in a connection coherency rule table, and in a source file formatting table, of information related to all of the possible configurations, each component obtaining a name that unambiguously identifies its position in the architecture, and a type from among several types (Active Components, Monitoring and Verification Blocks, Intermediate Blocks, System Blocks and Global Blocks).
- The instantiation of the components specified in the configuration definition file by the user-developer using a list of the components present, designated by their names and types and including parameters or invoking procedures, the configuration definition file comprising a file from which to select the components and their types and optional additional indications concerning the type of interface and the server involved in the configuration to be generated by the Configurator, and the storage of the corresponding information in an instance connection table.

- The topological connection of the instances based on the schema defined by the component and connection rule table (TCRC).
- The reading and storage of the HDL-type sources of the models instantiated, in order to extract the names and types of the interface signals from them.
- The wiring of the interface signals, at the level of each instance of the components by applying regular expressions, stored in the component and connection rule table (TCRC), based on the names of the signals constituting the wiring table (TCAB).
- The use of the instance connection table (TCINST), the wiring table (TCAB) and the formatting table (TFMT) to automatically generate the HDL-type (MGHDL) and HLL-type (MGHLL) source files of the global simulation model corresponding to the configuration specified by the configuration definition file (FCONF).

**[0084]** The topological phase proceeds in the following way:

- The explicit Components specified in the configuration file (FCONF) are first instantiated by the Configurator system and placed in the instance connection table (TCINST). This table contains the description of each instance (label, type), accompanied by a list of the components with which it is connected.
- Next, the Intermediate Blocks, explicitly specified in the ports of the "Active Components" in the configuration file, are instantiated and added to the instance connection table (TCINST).
- The connections between the active components instantiated are compared with the rules contained in the connection coherency table (TRCOH) in order to detect any incompatibilities between the ends of the connections between the components, and in that case to choose, and add to the instance connection table (TCINST), an adapter component (Intermediate Block) to be inserted into the connection in question.
- Next, the components of the Monitoring Block and Verification Block type are instantiated. The Configurator system analyzes the connections of the composite model components and searches for sharable interface adapters at the level of the common ports (see the models C_Port$_i$ in Fig. 3). If there is no suitable component with which to share the connection by observing the required signals, an interface adapter component with the properties specified

by the description will be instantiated by the configurator. The instance connection table (TCINST) is updated.

- Lastly, the "system block" components are instantiated and placed in the instance connection table (TCINST) for the components that need them.

[0085] In the Wiring phase, the Configurator system connects the interface signals at the level of each port by applying regular expressions based on the names of the signals as described in the definition of the port. These expressions are applied to the names of the signals extracted from the HDL-type descriptions of the components, so any changes in the HDL-type interfaces from one version of a model to another do not directly affect the specifications of the connections between the blocks. This phase results in the generation of the wiring table (TCAB), which associates the signals connected to one another with the unique name of the wire that connects them. A certain number of verifications are then possible, for example on the sourceless connections, or the connection of several outputs, or other connections.

[0086] The various steps in the wiring phase are the following:

- The Global Components and System Blocks are first wired to all of the components.
- Next, the signals between the other components are wired.
- After the wiring, an additional pass makes it possible to connect the signals of a component not connected in previous wiring phases to the corresponding signals of the system block, specially provided for forcing known and stable values that inhibit the unused ports. The signals in question of the system blocks carry a special marking (for example a special prefix in the name) so as not to be connected during the previous wiring phases.

[0087] In a source file generation phase, the Configurator system generates the HDL-type and HLL-type source files based on the content of the instance connection table (TCINST), the wiring table (TCAB) and the formatting table (TFMT), in order to automatically generate the HDL-type (MGHDL) and HLL-type (MGHLL) source files of the global simulation model corresponding to the configuration specified by the configuration file (FCONF).

[0088] The HDL-type source file contains a description of the HDL-type part (for example in VERILOG) of the global simulation model corresponding to the configuration file (FCONF). In particular, it contains the instantiation of all of the HDL-type models of the components mentioned in the configuration file, the intermediate blocks, System Blocks and Global Blocks, as well as all of the wires connecting these instances.

[0089] The HLL-type source file contains the program that corresponds to the instantiation of all the components having part of their models in an HLL-type language. In the case of object-oriented HLL languages, the code contains the constructors of HLL class objects with the corresponding parameters specified in the description of each component. A formatting table (TFMT) specific to each design makes it possible to generate the appropriate HLL-type code.

[0090] We will now describe the architecture description file (FDARCH) for a specific implementation of the configurator system (PROGCONF) in PERL language, chosen because of the direct manipulation of the regular expressions and the associative tables on several levels.

[0091] The architecture, represented in Fig. 1, is constituted by a processor (CPU) communicating through a bridge (BRIDGE) with a system memory (MEMORY) and input/outputs (I/O).

[0092] In order to facilitate its writing and manipulation, the file FDARCH represented has been divided into several parts, presented in Annexes A1 through A5. This file defines the architecture of the global simulation model corresponding to the generic diagram represented in Fig. 1. The HDL-type language generated is VERILOG and the high level language (HLL) generated is $C^{++}$.

[0093] The properties of the Components (type, HDL-type interfaces, ports, constructors of objects of the $C^{++}$ class, etc.) are described in several tables written in PERL language. The correspondence between these tables and the diagram illustrating the Configurator system of Fig 2 is represented in Fig. 6. In the description below, all of the notations beginning with "%..." correspond to hash tables. These tables can contain the description, either in the form of parameter sets or in the form of references to procedures that generate the required values.

[0094] The procedures are part of the description and are changed for each design. The typical example of their utilization is the generation of interconnections between the components. Often, a table corresponding to all of the possible interconnections would be too large and difficult to generate and maintain. A procedure, on the other hand, can be very compact.

[0095] The description of the rules for interconnecting the components (TCRC) contains global parameters common to all of the component types. It can exist in the form of at least one table. In the exemplary embodiment of Fig. 6, it is shown in the form of seven associated (hash) tables, three of which have the entries (%InstNameModuleMap, %SysConnectMap, %SysSpecMap), the names designating all of the possible models for the same component:

- The table "%InstNameModuleMap" (see 2 of A1), which appears in the file patent_config_defs.pm for describing the explicit Components. This table represents the rules for generating connections between the signals. It has the entries Connect and SelfConnect.
- The table "%SysConnectMap," which appears in the file patent_Sysconfig_defs.pm for the System Blocks.
- The table "%SysSpecMap," (see 4 of A2), which appears in the file patent_Sysconfig_defs.pm for the Intermediate Blocks.

[0096] The two hash tables %SysWrapMap and %SysPinConst (see 6 of A2) contain the system level connection rules. The hash table %Activity_TypeMap associates the name of a component with its type. The hash table %PortProbeMap has as entries the names of the HDL-type models used in the description of the components.

[0097] The Configurator system is controlled by at least one connection coherency rules table (TRCOH), which represents the rules for interconnecting between the components and inserting intermediate components. In the embodiment of Fig. 6, the connection coherency rules are distributed in the form of the following two tables:

- %HwfConnectivityMap (see. 2 of A2)
- %HwifAlternateMap

[0098] The Configurator system is controlled by at least one source file formatting table (TFMT), which represents the rules for generating instances of HLL-type objects. In the embodiment of Fig. 6, the rules for generating instances of HLL-type objects are distributed in the form of the following two tables:

- %moduleToCppClassMap (see 1 of A5)
- %classCppProtoMap (see. 2 of A5)

**[0099]** The role of each of these hash tables is described in detail below.

**[0100]** Among the entries defined for the tables %InstNameModuleMap, %SysConnectMap, %SysSpecMap, certain are mandatory:

- The entry "MatchExpr," whose value is the regular expression that makes it possible to accept a name (label) of the component in the configuration file, is used to define the variants of the positions authorized for the component.
- The entry "ExtrExpr," whose value is the regular expression that performs the extraction of the digital parameters from the name (label), is used for the instantiation of the Intermediate Blocks or the System Blocks or for generating the label of the implicit component.

**[0101]** Other parameters are optional and can be added to the description of the tables, but these parameters can only be used by the code provided with the description; among them:

- NumAdd is an auxiliary parameter serving as a modifier for the parameters extracted by ExtrExpr, used by the procedure GenDest (see below).
- BaseldExpr is a regular expression that makes it possible to find the active component corresponding to an intermediate or system component in case of ambiguity.

**[0102]** The following hash entries define the Type of the component, for example DUT (HDL-type model), XACTOR (transactor) or other types.

**[0103]** Each Component Type corresponds to a hash, in turn composed of the following entries:

- .The entry *ReqModule* contains the name of the HDL-type (VERILOG) module of the component and the name of the corresponding source file,
- The entry "Connect" is the definition of the method for selecting the signals that are part of a Port. This description is composed of a set of entries indexed by the name of the Port. Each Port name is associated with a table of regular expressions and a pointer to a procedure for connecting the signals, which controls the application of these expression to the names of the signals of the interface of the component.

[0104] The Configurator has several preprogrammed procedures, but others can be added and referenced. The preprogrammed procedures accept the modifiers:

- → E (exclusive – the signal will be used only once).
- → T (branch - the signal can be used by several destinations).
- → A special value "filter_out_wire_name" assigned to a signal or a group of signals makes it possible to eliminate any connection to the corresponding signals.

[0105] The entry "SelfConnect" -- similar to the entry "Connect" -- contains the definitions of the connections of the Port signals that can be connected between two instances of the same component. This definition facilitates the connection of one-way output ? input signals in the particular case of a head-to-tail connection.

– [0106] The entry "Port" provides the definition of all of the ports. It is a hash that associates the name of each port with a table that includes the following elements:
  - The logical name of the HDL-type component.
  - The name of the HDL-type module or a choice of modules.
  - The number of the port (useful for generating VERILOG and $C^{++}$ code).

– [0107] The entry "GenDest (see 1 of A3) is a pointer to the procedure that, for a component designated by a label and type, generates, for each port, the label of the component to which it is connected. The referenced procedure must imperatively be specified by the user; this procedure is also used to control the coherency of the requested configuration.

– [0108] The entry "SysConn" is a hash that associates each HDL-type module contained in the model with a pointer to the procedure for selecting and naming the System Blocks. The referenced procedure must imperatively be specified by the user.

– [0109] The entry "GenHDLInstParam" is a pointer to the procedure that generates the additional parameters required by the HDL-type simulator for the code generated, which instantiates the HDL-type part of the component.

– [0110] The entry "CfgCpp" defines the parameters of the constructor of the object for the $C^{++}$ identification code of the Configuration, generated automatically, which serves as a filter for the selection of the Components by the Configurator system.

- **[0111]** The first parameter is the name of the class of the C$^{++}$ object; it is a predefined name associated with that of the HLD-type model. It is followed by the tables that correspond to groups of parameters typically associated with each port of the component. The keyword "Own" indicates an elementary model component.
- **[0112]** Next comes the name of the HDL-type component referenced in the Port part and the type identifier of the component, for example DUT, model of the circuit under design, XACTOR, or transactor.
- **[0113]** For the composite model blocks, the parameter part is richer.
  - → **[0114]** It contains the keyword "Src" for the active components or "Mon" for the Verification and Monitoring components.
  - → **[0115]** Next come the parameters of the interface adapter component, comprising:
    - the name of the HDL-type component.
    - the identifier of the component type and its class followed by the name of the corresponding port.

    **[0116]** The latter parameters are specific to each port of the composite block.

**[0117]** In the C$^{++}$ code generated for a composite model, the parameters transmitted to the constructor of a class correspond to the pointers to the instances of the interface adapters.

**[0118]** All of these parameters serve to guide the generator of the C$^{++}$ code, which combines them with the rules contained in the hash table "%classCppProtoMap":

– The entry "ProbeConnect" associates each port of a Monitor or Verifier component with the C$^{++}$ class that is acceptable as an interface adapter. Thus, the Configurator is capable of optimizing the code by using the same interface adapter for several components (composite model).

– The entry "SysWrap" is a hash that, for each HDL-type part of a component, defines the regular expression to be used for setting the remaining unconnected signals to a known state after the wiring between the components.

**[0119]** The algorithm for connecting two components is the following:

- The Configurator system first tries direct connection.
- If this proves impossible based on the first hash table %HwfConnectivityMap, the modules specified in %HwifAlternateMap are retrieved in order to be placed in the component. The optional modules are indicated by the character ">" at the beginning of their name in the specification of a component.
- In the end, if no module works, the Configurator system returns to the hash %HwfConnectivityMap to choose the intermediate blocks to be placed between the components.
- If no suitable block is found, an error is signaled.

**[0120]** The user can influence the Configurator system's choice by explicitly specifying in the configuration file the intermediate blocks to be connected to a component.

**[0121]** The table "%HwfConnectivityMap" allows for both the Verification of connectivity and the specification of intermediate Blocks to be inserted in order to connect two Components whose parts are not directly connectable.

**[0122]** In order for two blocks to be able to be connected to each other, it is necessary for %HwfConnectivityMap to contain entries that describe the properties of this connection.

- Each component is associated with the component to which it can be connected.
- A direct connection is indicated by the value "Direct."
- For connections requiring an intermediate Block, the name of this block is indicated.
- For certain blocks, the connectivity can be specified differently for each port of the block. This serves to eliminate the ambiguity in the connections of a block. A typical utilization involves the insertion of two intermediate blocks connected head-to-tail between two explicit components.

**[0123]** It is emphasized that the connectivity in this case is expressed globally, without specifying the names of signals to be connected.

**[0124]** The Components described in an HLL-type language can have several possible implementations of the HDL-type modules A_Port, at the level of each port i (see Fig. 3). The hash table "%HwifAlternateMap" specifies the choice that is possible for each HDL-type module of a component. The configurator system uses this information in order to insert the intermediate blocks only when strictly necessary.

**[0125]** The hash %SysWrapMap defines, for each implementation of a component, its System Block. The entries are names of HDL-type modules associated with the name of the system block and its HDL-type implementation. This information is used by the Configurator system to instantiate the appropriate system block from specifications contained in the hash %SysConnectMap (see 3 of A2).

**[0126]** The hash %SysPinConst associates, with each System Block, the connections of certain signals of its interface to the names of the corresponding signals (wires) at the system level. These connections are independent of the configurations and are not governed by the regular expressions that are applicable during the wiring of the other signals.

**[0127]** Typically, the signals in question are clock signals, reset signals, etc. The referenced names are declared in an HDL-type code that is specific for each model under design, and are provided by the user (in the form of a variable or a file) with the descriptions of the components and the specific connection rules.

**[0128]** This specific HDL-type code is systematically included at the beginning of the HDL-type code generated by the Configurator system.

**[0129]** The procedure "&gen_sys_VlogInstParameter" is a procedure that generates the instantiation parameters for the generator of the HDL-type code from the label and of the type of each Component.

**[0130]** The hash %Activity_TypeMap (see 1 of A1) associates the name of a component with its type, specified by a keyword:

- "actor" for the active components.
- "spectator" for the Monitoring blocks.
- "checker" for the Verification blocks.
- "helper" for the System blocks or the Intermediate blocks in case of an indirect connection.

**[0131]** The table "%PortProbeMap" is a hash whose entries are the names of the HDL-type models used in the description of the components. The values are in turn hashes that associate, with each port name, a table containing the name of a block that can serve as a Probe for observing the signals of this port and the name of the corresponding $C^{++}$ class. This hash informs the Configurator system of the type of component capable of observing the signals of a given port and the type of C++ API that can be used to analyze the control and data flow observed in this port.

**[0132]** The procedure &GenDest accepts as input the label of the source component, its type and its port. The result returned is a hash that associates the label of the connected target component with its port.

**[0133]** In order to control the coherency of the configuration generated no matter what the order in which the Components are handled, the Configurator system calls the procedure &GenDest with the parameters obtained for the target component used as a source and verifies that the result actually corresponds to the initial source component and port.

**[0134]** The Configurator system calls the procedure &GenDest for each component contained in the configuration file, running through all of these ports.

**[0135]** The hash returned by &GenDest can contain several components with their respective ports. This is particularly the case for connections of the "bus" type; &GenDest returns all of the components connected by the busses except the one that is given as an argument. The Configurator system automatically inserts the necessary "bus" component.

**[0136]** The hash &moduleToCppClassMap is a search accelerator that makes it possible to retrieve the C++ class from the name of the HDL-type module; it only applies to elementary Models (see 1.14).

**[0137]** The hash %classCppProtoMap describes the generation of the constructors of C++ class objects. Each entry corresponds to the name of a C++ class. Each description is a hash composed of four entries:

- The entry "Prea" corresponds to the generation of the first part of the constructor (class, name of the instance, first parameters).
- The entry "Iter" corresponds to the iterative part of the parameters of the constructor. This rule is applied for each element contained in the entry CfgCpp of the description of the component.
- The entry "Post" corresponds to the last part of the constructor (last parameters, closing parentheses).
- The entry "Idle" is a rule that substitutes for the rule "Iter" for nonexistent connections (partial configurations). Typically, a null pointer or null string is generated.

**[0138]** Each entry is a character string containing predetermined special selectors, replaced by the values calculated by the Configurator system. These selectors have the general form #<Src¦Dst¦Act><Item#, where "Src," "Dst" and "Act" indicate the instance to be used, the latter being transmitted as a parameter to a constructor of an HLL object.

- "Src" indicates the current instance.
- "Dst" indicates the instance connected to the other end of the port.
- "Act:" indicates the composite instance corresponding to the Active Component containing the observation port.

**[0139]** <Item> is a keyword indicating the selection:

- "Inst" for the instance name cpp.
- "Iid" for the component label corresponding to Inst.
- "Ict" for the component type (DUT, VERIFIER, XACTOR, Monitor, etc.).
- "Port" for the name of the port.

**[0140]** The two variables "$cpp_preamble" (see 3ofn A5) and "$cpp_postamble" (see 4 of A5) respectively contain the beginning and the end of the $C^{++}$ program text generated by the Configurator system. The text generated from the table %classCppProtoMap is inserted in the middle.

**[0141]** The two variables "$top_verilog_preamble" and "$top_verilog_postamble" respectively contain the beginning and the end of the text of the instance "top" of the HDL-type (VERILOG) description of the configuration. Typically, they are the instantiations of clock blocks and other system level blocks.

**[0142]** The description that follows is a simple example of the generation of a Configuration through the implementation of the Configurator system, which is illustrated in a simplified case based on the architecture of a System constituted by a bridge (BRIDGE) that connects a central processor (CPU), a Memory, and an input/output card (I/O). The Architecture of the System is represented in Fig. 1.

**[0143]** In this example, the CPU and the BRIDGE can be modeled, either by a VERILOG model of the design (DUT, DUT_Core), or by a $C^{++}$ composite model (XACTOR).

**[0144]** The interface between the CPU and the BRIDGE is called FBUS. Several variants of this interface are considered to correspond to successive stages of development.

**[0145]** The global block Clock handles the distribution of the clock signals and system signals globally for all of the Components.

**[0146]** The Configurator system relies on the description of the HDL-type Interfaces of the Components. Simple interfaces, written in VERILOG language, are proposed below as examples for each of the Components. The Configurator system analyzes only this part of the source files contained in the library of HDL-type source files (BFSHDL). They belong to a methodology for progressively handling variants of models as they become available.

**[0147]** The following HDL-type description of the Interfaces has been reduced to only what is necessary to illustrate the capabilities and the operation of the configurator. In a real situation, the behavioral descriptions are included, as in the case of the model of the clock (the module CLOCK).

**[0148]** The description is as follows:

– CPU :

a) The model DUT of the component CPU (Fig. 7a) has a port FBUS for connecting with the BRIDGE, as well as a set of System signals, defined below.

```
module cpu (
        XXadr_dat,
        XXreq,
        XXack,
        //
        clk,
        reset,
        powergood
        );
inout   [63:0] XXadr_dat;
inout   [3:0] XXreq;
inout   [3:0] XXack;

input   clk;
input   reset;
input   powergood;

endmodule
```

b) The model XACTOR (Fig. 7b) consists in an abstract $C^{++}$ model whose HDL-type instance is constituted by that of the Port named fbus_p. In this example, it is assumed that the Port FBUS has a variant named FBUSA wherein the two-way signals adr_dat have been separated into pairs of one-way in (inXXadr_dat) and out (outXXadr_dat) signals.

```
module fbus_p (
        inXXadr_dat,
        outXXadr_dat,
        inXXreq,
        outXXreq,
        inXXack,
        outXXack,
        //
        clk,
        reset,
        powergood
);

input   [63:0] inXXadr_dat;
output  [63:0] outXXadr_dat;
input   [3:0] inXXreq;
output  [3:0] outXXreq;
input   [3:0] inXXack;
output  [3:0] outXXack;

input   clk;
input   reset;
input   powergood;

endmodule
```

c) The model MONITOR of the component CPU is a composite model that allows the Monitoring of the interface FBUS. In the case where there is no port adapter fbus_p in the configuration, a probe containing the instance fbus_probe will be generated (Fig. 7c).

```
module fbus_probe (
        XXadr_dat,
          XXreq,
          XXack
          //
          clk,
          reset,
          powergood
          );
input     [63:0] XXadr_dat;
input     [3:0] XXreq;
input     [3:0] XXack;

input     clk;
input     reset;
input     powergood;

endmodule
```

- Intermediate adaptation blocks FBUS:

**[0149]** These blocks implement the interface adaptation at the FBUS level in the case where point-to-point mapping of the signals is impossible. The port FBUSA is the variant of FBUS in which the two-way signals have been separated into pairs of one-way incoming (in) and outgoing (out) signals.

**[0150]** The model fbus_xchg (Fig. 7d) implements the adaptation between the interface FBUS of the port fbus_p and that of a port FBUSA; the width of the interface can be parameterized so as to accommodate potential technological evolutions.

```
module fbus_xchg (
          XXadr_dat,
          XXreq,
          XXack,
          //
          inXXadr_dat,
          outXXadr_dat,
          inXXreq,
          outXXreq,
          inXXack,
          outXXack,
);
inout     [63:0] XXadr_dat;
inout     [3:0] XXreq;
inout     [3:0] XXack;
          //
input     [63:0] inXXadr_dat;
output    [63:0] outXXadr_dat;
input     [3:0] inXXreq;
```

```
output [3:0] outXXreq;
input  [3:0] inXXack;
output [3:0] outXXack;

// model body

endmodule
```

– BRIDGE:

a) The model DUT of the component BRIDGE (Fig. 7e) has the three respective ports to the models CPU, MEM and IO as well as the System signals delivered by the dedicated system block SYS_BRIDGE and by the global block CLOCK.

```
module bridge (
        XXadr_dat,
        XXreq,
        XXack,
        YYadr,
        YYdata,
        ZZdata,
        ZZreq,
        ZZack,
        YYctrl,
        clk_2xp,
        clk_2xn,
        clkp,
        clkn,
        reset,
        powergood
        );
inout  [63:0] XXadr_dat;
inout  [3:0] XXack;
inout  [3:0] XXreq;

output [31:0] YYadr;
inout  [63:0] YYdata;
output [2:0] YYctrl;

inout  [15:0] ZZdata;
input  [1:0] ZZack;
output [1:0] ZZreq;

input  clk_2xp;
input  clk_2xn;
input  clkp;
input  clkn;
input  reset;
input  powergood;
```

```
endmodule
```

b) The model BRIDGE_CORE (Fig. 7f) is the variant DUT_CORE of the component BRIDGE having an interface FBUSA. This model reflects the situation in which the model of the FBUS interface controller is not available.

```
module bridge_core (
        inXXadr_dat,
        outXXadr_dat,
        inXXreq,
        outXXreq,
        inXXack,
        outXXack,
        //
        YYadr,
        YYdata,
        YYctrl,
        //
        ZZdata,
        ZZreq,
        ZZack,
        //
        clk_2xp,
        clk_2xn,
        clkp,
        clkn,
        reset,
        powergood
        );
input  [63:0] inXXadr_dat;
output [63:0] outXXadr_dat;
input  [3:0] inXXreq;
output [3:0] outXXreq;
input  [3:0] inXXack;
output [3:0] outXXack;

output [31:0] YYadr;
inout  [63:0] YYdata;
output [2:0] YYctrl;

inout  [15:0] ZZdata;
input  [1:0] ZZack;
output [1:0] ZZreq;

input  clk_2xp;
input  clk_2xn;
input  clkp;
input  clkn;
input   reset;
input   powergood;


endmodule
```

c) The model sys_bridge (Fig. 7g) is the system block dedicated to the System signals of the Bridge model; it is fed directly by the block CLOCK.

```
module sys_bridge(
                clk_2xp,
                clk_2xn,
                clkp,
                clkn,
                reset,
                powergood,
                //
                sys_CLK_2X,
                sys_CLK,
                sys_RESET,
                sys_POWER_GOOD
                );

output          clk_2xp;
output          clk_2xn;
output          clkp;
output          clkn;
input           sys_CLK_2X;
input           sys_CLK;
input           sys_RESET;
input           sys_POWER_GOOD;
output          reset;
output          powergood;

clk_2xp = sys_CLK_2X;
clk_2xn = ~sys_CLK_2X;
clkp    = sys_CLK;
clkn    = ~sys_CLK;
reset,  = sys_RESET;
powergood = sys_POWER_GOOD;

endmodule
```

d) The model XACTOR (Fig. 7h) is an abstract $C^{++}$ model whose HDL-type instance is constituted by that of the various ports, respectively fbus_p, cmem_p, and cio_p, to CPU, MEM and IO.

```
module cmem_p (
                YYadr,
                YYdata,
                YYctrl,
                clk,
                reset,
                powergood
                );
        output [31:0] YYadr;
        inout  [63:0] YYdata;
        output [2:0] YYctrl;

        input  clk;
        input  reset;
        input  powergood;

        endmodule

        module cio_p (
                ZZdata,
                ZZreq,
                ZZack,
                //
                clk,
                reset,
                powergood
                );
        inout   [15:0] ZZdata;
        input  [1:0] ZZack;
        output [1:0] ZZreq;

        input  clk;
        input  reset;
        input  powergood;

        endmodule
```

– MEMORY :

The model DUT of the component MEMORY (Fig. 7i) has the respective ports to the models BRIDGE and CLOCK.

model DUT:

```
        module cmem (
                YYadr,
                YYdata,
                YYctrl,
                clk,
                reset,
                powergood
                );
```

```
input   [31:0] YYadr;
inout   [63:0] YYdata;
input   [2:0] YYctrl;

input   clk;
input   reset;
input   powergood;

endmodule
```

– I/O:

The model DUT of the component I/O (Fig. 7j) has the respective ports to the models BRIDGE and CLOCK.

model DUT:

```
module cio (
        ZZdata,
        ZZreq,
        ZZack,

        clk,
        reset,
        powergood
        );
inout    [15:0] ZZdata;
output [1:0] ZZack;
input   [1:0] ZZreq;

input   clk;
input   reset;
input   powergood;

endmodule
```

– Global System Block:

This model (Fig. 7k) handles the delivery of the System signals that feed the system blocks dedicated to each component.

```
module Clock (
                        sys_POWER_GOOD,
                        sys_RESET,
                        sys_CLK,
                        sys_CLK_2X
                        );
output sys_POWER_GOOD;
output sys_RESET;
output sys_CLK;
output sys_CLK_2X;

parameter HALF_PERIOD = 75; // 7.5ns, 66MHz
parameter RESET_DELAY = 10000000000 ; // 1.0ms
parameter POWER_GOOD_DELAY = 5000000000 ; // 0.5ms
initial begin;
 sys_POWER_GOOD = 0;
sys_ RESET = 0;
 sys_CLK = 0;
 sys_CLK_2X = 0;
 #POWER_GOOD_DELAY sys_POWER_GOOD = 1;
 #RESET_DELAY sys_RESET = 1;
end

always begin
 #HALF_PERIOD sys_CLK = ~sys_CLK;
end
always @(posedge sys_CLK_2X) begin
  sys_CLK = ~sys_CLK;
end

endmodule
```

**[0151]** Definition of the configurations:

**[0152]** The configuration variants mark the various verification stages based on the availability of the models; the Configurator system generates the configurations unambiguously from topological information contained in the configuration file.

**[0153]** The first configuration is defined by the configuration file 1 below, represented in Fig. 8a; the models DUT of the CPU and the BRIDGE are not available.

Configuration file 1:

```
CPU_0XACTOR
CPU_0MONITOR
BRIDGE_0   XACTOR
CMEM_0     DUT
CIO_0      DUT
```

The XACTOR CPU shares the port fbus_p with the MONITOR.

[0154] The second configuration is defined by the configuration file 2 below, represented in Fig. 8b.

[0155] The BRIDGE XACTOR is connected to the CPU XACTOR by an FBUSA-type interface specified in the configuration file by the attribute FBUSA=FBUS_xchg; the Configurator system automatically inserts the second adapter fbus_xchg 102.

[0156] Configuration file 2:

```
CPU_0       XACTOR    FBUSA=FBUS_xchg
CPU_0       MONITOR
BRIDGE_0    XACTOR
CMEM_0      DUT
CIO_0       DUT
```

[0157] The XACTOR CPU shares the port fbus_p with the Monitor.

[0158] The third configuration is defined by the configuration file 3 below, represented in Fig. 8c; the BRIDGE DUT_CORE is connected to the CPU XACTOR by an FBUSA-type interface specified in the configuration file by the attribute FBUSA=FBUS_xchg; the Configurator system automatically inserts the second adapter fbus_xchg 102.

[0159] Configuration file 3:

```
CPU_0       XACTOR    FBUSA=FBUS_xchg
CPU_0       MONITOR
BRIDGE_0    DUT_CORE
CMEM_0      DUT
CIO_0       DUT
```

[0160] The XACTOR CPU shares the port fbus_p with the MONITOR.

[0161] The fourth configuration is defined by the configuration file 4 below, represented in Fig. 8d; the model CPU DUT is connected to the model BRIDGE XACTOR. The Configurator system automatically inserts an interface adapter fbus for implementing the connection between the CPU DUT and the BRIDGE XAXTOR without any explicit mention of this block in the configuration file.

**[0162]** Configuration file 4:

|  |  |
|---|---|
| CPU_0 | DUT |
| CPU_0 | MONITOR |
| BRIDGE_0 | XACTOR |
| CMEM_0 | DUT |
| CIO_0 | DUT |

**[0163]** The Configurator automatically instantiates the Probe fbus_probe in order to allow the observation of the interface FBUS by the Monitor.

**[0164]** The fifth configuration is defined by the configuration file 5 below, represented in Fig. 8e; all of the models DUT are available.

**[0165]** Configuration file 5:

|  |  |
|---|---|
| CPU_0 | DUT |
| CPU_0 | MONITOR |
| BRIDGE_0 | DUT |
| CMEM_0 | DUT |
| CIO_0 | DUT |

**[0166]** The Configurator system automatically instantiates the Probe fbus_probe in order to allow the observation of the interface FBUS by the Monitor.

**[0167]** The sixth configuration is defined by the configuration file 6 below, represented in Fig. 8f; it is a multi-server Configuration; Config1 is distributed between the 2 servers SERVER1 and SERVER2, the cutoff occurring at the level of the interface FBUS.

**[0168]** Configuration file 6:

|  |  |  |
|---|---|---|
| CPU_0 | XACTOR | SERVER1 |
| CPU_0 | MONITOR | SERVER1 |
| BRIDGE_0 | XACTOR | SERVER2 |
| CMEM_0 | DUT | SERVER2 |
| CIO_0 | DUT | SERVER2 |

[0169] The connectivity rules that make it possible to generate the connectivity tables of any configuration for this example are described in PERL language in Annexes A1 through A5.

[0170] Annex A1 describes, in the case of the example chosen, the topological rules linked to the active components to be used by the Configurator system.

[0171] Annex A2 describes the topological rules linked to the system Blocks and the Intermediate blocks to be used by the Configurator system.

[0172] Annex A3 describes the procedures for connecting the Components to be used by the Configurator system.

[0173] Annex A4 describes the code for generating the file in HDL-type language (Verilog_top.v) for connecting the components to be used by the Configurator system.

[0174] Annex A5 describes the code for generating $C^{++}$ Classes.

[0175] Annexes A6 and A7 describe the result files generated by the Configurator system, respectively in a low level HDL-type language (VERILOG) and in a high level language ($C^{++}$), in order to create the model of the configuration defined by the third configuration corresponding to Fig. 8c. The configurator can thus, with the user-developer, create its own model in accordance with the configuration variant chosen.

[0176] It would have been possible, in the same way, to illustrate the other variants of embodiment of the model (Figs 8a; 8b and 8d through 8f) that could be processed by the Configurator system in order to produce the corresponding result files constituting the simulation models of the system architecture envisioned for the integrated circuit under development.

[0177] Fig. 2 represents the various means used by a computer system 40 implementing the method of the invention.

[0178] The computer 40 includes a central processor 41 comprising data processing circuits 42 (an arithmetic and logic unit ALU and memories of associated programs, hereinafter referred to globally as ALU, for convenience), and includes various memories, programs or working data files associated with the central processor 41.

[0179] The central processor 41 includes a memory 51 containing the component and connection rule table (TCRC), the connection coherency rule table (TRCOH) and the source file formatting table (TFMT). These tables are created from the architecture description file (FDARCH).

[0180] The memory 51 also contains the necessary programs or intelligent engines PROGCONF for constituting the Configurator system, which are usable by the ALU 42.

[0181] A memory 53 contains the instance connection table (TCINST) representing the instances of the components and their mutual interconnections required by the Configuration defined in the configuration file FCONF and conforming to the rules contained in the tables TCRC and TRCOH.

[0182] The memory 53 also contains the wiring table (TCAB) representing the physical connections (wires) between the HDL-type parts of the components instantiated from the HDL-type source file library (BFSHDL).

[0183] This results in the files MGHDL and MGHLL, which respectively represent the HDL-type (VERILOG or VHDL) and HLL-type ($C^{++}$) source files of the global simulation model generated by the Configurator system, a model that allows the co-simulation of an ASIC circuit in its verification environment.

[0184] The implementation of the method of the invention will now be explained.

[0185] The method for automatically generating, by means of the computer or data processing system 40, a simulation model of a configuration of models of given components, mutually linked by interworking connections so as to create a global simulation model MGHDL/MGHLL of an ASIC circuit in its verification environment, includes the following steps (see Fig. 2a):

- an operator, possibly aided or even replaced by a computer, generates the architecture description file (FDARCH) describing the rules for interconnection between the various components, the models being capable of being used to model each of these components and the formatting rules in order to generate the source files of the resulting model.
- an operator, possibly aided or even replaced by a computer, generates the source file library BFSHDL from HDL-type parts of models of respective components capable of being used in a configuration in conformity with the content of the file FDARCH.

- the operator generates the desired configuration file FCONF, which identifies the components and the desired models (see Fig. 2b).
- The data processing system reads the architecture description file (FDARCH) and generates and places in memory the tables TCRC, TRCOH and TFMT from FDARCH (see Fig. 2b).
- The various tables TCRC, TRCOH and TFMT having been made accessible to the data processing system 40 (see Fig. 2c),
- the data processing system 40 reads the desired configuration file FCONF in order to identify the components and their required models (see Fig. 2c);
- it reads the component and connection rule table in order to instantiate the required components and place them in the instance connection table TCINST (see Fig. 2c).
- It reads the connection coherency rule table (TRCOH) in order to verify the physical connectivity between models and, if necessary, inserts intermediate components.
- It reads, in the library BFSHDL, the source files corresponding to the HDL-type parts of the component models instantiated in the table TCINST in order to generate the table TCAB of physical connections (wires) between the component models (see Fig. 2c).
- It applies the formatting rules from the table TFMT to the contents of the tables TCINST and TCAB, and it generates the final source files MGHDL/MGHLL of the global simulation model corresponding to the configuration specified by the configuration file (FCONF) (see Fig. 2d).

[0186] The input file FCONC corresponds to a description of the theoretical diagram of the type in Fig. 1, but without the models, which are added during the generation of the intermediate tables TCINST, TCAB describing the configuration and connections, in order to provide a global simulation model that is actually operational and that, moreover, allows the observation of particular signals.

[0187] The various tables above can be pre-stored in the computer 40 or can even be stored in an external memory that one connects to the computer 40, possibly through a data transmission network, in order to make them accessible.

[0188] The ALU 42 has all of the basic data required by the table tools TCRC, TRCOH, TFMT in order for the program PROGCONF associated with the Configurator system to control, in the ALU 42, the generation of the final files MGHDL/MGHLL (see Fig. 2d).

[0189] The intermediate table TCAB describing the physical connections generated from an HDL_type description, is, in this example, subjected to a verification in order to detect any obvious errors, such as the existence of unconnected inputs or outputs or even outputs connected in parallel, when there should be conventional outputs of the so-called "totem-pole" type, as opposed to those of the open collector or tri-state type.

[0190] In case of a modification of the initial file FCONF, the intermediate tables TCINST, TCAB and the final files MGHDL/MGHLL can thus be redefined from the files of models and from the file FDARCH. This avoids any risk of error.

[0191] In this example, the ALU 42 generates a global co-simulation model by associating, with at least one functional block, several functional specification models written in programming languages of different levels, in this case of the HDL and $C^{++}$ types.

[0192] It is understood that the present invention can be implemented in other specific forms without going beyond its scope of application as claimed. Consequently, the present detailed description should be considered to be a simple illustration of a particular case within the context of the invention, and can therefore be modified without going beyond the scope defined by the attached claims.

## ANNEX: Description of th  Configurator Code

### A1 - Topological Rules Linked to the Activ  Components

```
#! /usr/triton/bin/perl
################################################################
##############
#
#      Copyright (c) 2000 BULL - Worldwide Information Systems
#                        All rights reserved
#
#   Module name   :  patent_config_defs.pm
#   Author        :  Andrzej Wozniak
#
################################################################
##############

###
%Activity_TypeMap =
  (
   "XACTOR"             => "actor",
   "DUT"                => "actor",
   "DUT_CORE"           => "actor",
   "MONITOR"            => "spectator",
   "IND_CON"            => "helper",
   "SYS_CON"            => "helper",
   );
```

1 of A1

```
###################
%InstNameModuleMap =
  ###################
  (
   #######
  "CPU" =>
   #######
   { "MatchExpr" => "^(CPU)(?:_[0-3])\$" ,
     "ExtrExpr"  => "^CPU_([0-3])\$" ,
     ####
     "DUT"   =>
     { "ReqModule" => { "cpu" => "cpu_model.v",
              },

       "Connect" => { FBUS => [\&subst_infix,  ["(XX.*)",
""],],
              },
       "Port"    => { FBUS => [CPU_a, cpu, 0], },
       "genDest" => \&genDest,
       "SysConn" => { CPU_a => \@std_glob_con,
              },
```

2 of A1

List of the accepted CPU labels

Regular connection expressions

```
        "CfgCpp"   => [CPU_Dut,  [ ['Own', [CPU_a , DUT, ],
['None'], ], ], ],

      },
```

**CPU_a** = logical name
**cpu** = name of the Verilog instance generated
**0** = Port no.
**\&genDest** = defines the procedure that generates the genDest

```
     "XACTOR"  =>
     { "ReqModule" => { "fbus_p" => "fbus_port.v",
                 },
```

Same connection name on both ends

```
       "Connect"      => { "FBUSA"   => [\&subst_infix,
["in(.*)", "aa_con"], ["out(.*)", "bb_con"],],
                   },
```

Head-to-tail connection

```
       "SelfConnect" => { "FBUSA"   => [\&subst_infix,
["out(.*)", "aa_con"], ["in(.*)", "bb_con"],],
                   },
       "Port"     => { FBUSA => [FBUS_p, fbus_p, 0], },
       "genDest" => \&genDest,
       "SysConn" => { FBUS_p => \@std_glob_con,
               },
       "CfgCpp"  => [CPU_Xactor, [ ['Src', [FBUS_p ,
FBUS_type, Fbus_hwif], [FBUS] ] ] ],

     },
     "MONITOR"  =>
     { "ReqModule" => { "fbus_p" => "fbus_port.v",
                 },

       "Connect"      => { "FBUS"   => [\&subst_infix,
["in(.*)", "aa_con"], ["out(.*)", "bb_con"],],
                   },
       "SelfConnect" => { "FBUS"   => [\&subst_infix,
["out(.*)", "aa_con"], ["in(.*)", "bb_con"],],
                   },
       "Port"     => { FBUS => [FBUS_p, fbus_p, 0], },
       "genDest" => \&genDest,
       "SysConn" => { FBUS_p => \@std_glob_con,
               },
       "ProbeConnect" => { FBUS => Fbus_hwif,
                  },
      "CfgCpp"  => [CPU_Monitor, [ ['Mon', [FBUS_p ,
FBUS_type, Fbus_hwif], [FBUS],],],],

    },
     ####
   },### CPU
   ######
   "BRIDGE" =>
   ######
   {
     "MatchExpr" => "^(BRIDGE)(?:_[0-1])\$" ,
     "ExtrExpr"  => "^BRIDGE_([0-1])\$" ,
     ####
     "DUT"   =>
     { "ReqModule" => {"bridge" => "bridge_x.v",
             },
       "Connect" => { FBUS => [\&subst_infix,  ["(XX.*)",
""],],
                     CMEM => [\&subst_infix,  ["(YY.*)", ""],],
                     CIO  => [\&subst_infix,  ["(ZZ.*)", ""],],
            },

       "Port"     => {  FBUS => [BRD, bridge, 0],
                      CMEM => [BRD, bridge, 1],
                      CIO  => [BRD, bridge, 2],
             },

       "genDest" =>  \&genDest,
       "SysConn" => { BRD => [\&getSpecSysInst, "E",
[\&subst_infix,  ["(.*)", ""],],],
            },
       "CfgCpp"  => [Bridge_Dut, [ ['Own', [BRD, DUT, ],
['None'], ],
                               ],
                  ],
     },
      ####
     "DUT_CORE"  =>
     { "ReqModule" => {"bridge_core" => "bridge_core.v",
             },
       "Connect" => { FBUSA => [\&subst_infix,  ["(XX.*)",
""],["in(.*)", "aa_con"], ["out(.*)", "bb_con"],],
                     CMEM  => [\&subst_infix,  ["(YY.*)", ""],],
                     CIO   => [\&subst_infix,  ["(ZZ.*)", ""],],
            },

      "SelfConnect" => { "FBUSA"  => [\&subst_infix,
["out(.*)", "aa_con"], ["in(.*)", "bb_con"],],
                  },

       "Port"     => {  FBUSA => [BRD, bridge_core, 0],
                  CMEM   => [BRD, bridge_core, 1],
                  CIO    => [BRD, bridge_core, 2],
              },

        "genDest" =>  \&genDest,
        "SysConn" => { BRD => [\&getSpecSysInst, "E",
[\&subst_infix,  ["(.*)", ""],],],
              },
        "CfgCpp"  => [Bridge_Dut, [ ['Own', [BRD, DUT_CORE,
], ['None'], ],
                            ],
                ],
      },
       ####
     "XACTOR"  =>
     { "ReqModule" => { "fbus_p" => "fbus_port.v",
                  "cmem_p" => "cmem_port.v",
                  "cio_p"  => "cio_port.v",
                },
       "Connect" => { FBUSA => [\&subst_infix,  ["in(.*)",
"aa_con"], ["out(.*)", "bb_con"],],
                CMEM   => [\&subst_infix,  ["(YY.*)", ""],],
                CIO    => [\&subst_infix,  ["(ZZ.*)", ""],],
              },

       "SelfConnect" => { "FBUSA"  => [\&subst_infix,
["out(.*)", "aa_con"], ["in(.*)", "bb_con"],],
                  },

       "Port"    => {  FBUSA => [FBUS_p, fbus_p, 0],
                 CMEM   => [CMEM_p, cmem_p, 1],
                 CIO    => [CIO_p,   cio_p, 2],
               },

       "genDest" =>  \&genDest,
       "SysConn" => { FBUS_p => \@std_glob_con,
                CMEM_p => \@std_glob_con,
                CIO_p  => \@std_glob_con,
              },
       "CfgCpp"  => [Bridge_Xactor, [ ['Src', [FBUS_p,
FBUS_type, Fbus_hwif], [FBUS] ],
                         ['Src', [CMEM_p, CMEM_type,
Cmem_hwif], [CMEM] ],
                         ['Src', [CIO_p,  CIO_type,
Cio_hwif], [CIO] ],
                         ],
               ],
     },
     },
   ####
   ######
```

```
"CIO" =>
######
{
  "MatchExpr" => "^(CIO)(?:_[0-1])\$" ,
  "ExtrExpr"  => "^CIO_([0-1])\$" ,
  ####
  "DUT"   =>
  { "ReqModule" => { "cio" => "cio_model.v",
                   },
    "Connect" => { CIO  => [\&subst_infix,  ["(ZZ.*)",
""],],
                 },

    "Port"     => {  CIO  => [CIOD,   cio, 0],
                  },

    "genDest" =>  \&genDest,
    "SysConn" => { CIOD => \@std_glob_con,
                 },
    "CfgCpp"  => [Cio_Dut, [ ['Own', ['Src', [CIOD,
DUT,], [None] ],],
                       ],
                  ],
  },
    ####
 "XACTOR" =>
 { "ReqModule" => { "cio_p" => "cio_model.v",
                  },
   "Connect" => { CIO  => [\&subst_infix,  ["(ZZ.*)",
""],],
               },

   "Port"     => {  CIO  => [CIOD,   cio, 0],
                   },

   "genDest" =>  \&genDest,
   "SysConn" => { CIOD => \@std_glob_con,
                },

   "CfgCpp"  => [Cio_Dut, [ ['Own', ['Src', [CIOD,
DUT,], [None] ],],
                        ],
                   ],
    },
    },
######
"CMEM" =>
######
{
  "MatchExpr" => "^(CMEM)(?:_[0-1])\$" ,
  "ExtrExpr"  => "^CMEM_([0-1])\$" ,
      ####
      "DUT"   =>
      { "ReqModule" => { "cmem" => "cmem_model.v",
                  },
        "Connect" => { CMEM   => [\&subst_infix,   ["(ZZ.*)",
"" ], ],
                },

        "Port"     => {   CMEM   => [CMEMD,    cmem,  0],
                 },

        "genDest" =>   \&genDest,
        "SysConn" => { CMEMD => \@std_glob_con,
                },

        "CfgCpp"   => [Cmem_Dut,  [  ['Own',  ['Src',  [CMEMD,
DUT,],  [None]  ],],
                      ],
                 ],
      },
        ####
     "XACTOR" =>
     { "ReqModule" => { "cmem_p" => "cmem_model.v",
                 },
       "Connect" => { CMEM   => [\&subst_infix,   ["(ZZ.*)",
"" ], ],
               },

       "Port"     => {   CMEM   => [CMEMD,    cmem,  0],
                   },

       "genDest" =>   \&genDest,
       "SysConn" => { CMEMD => \@std_glob_con,
               },
       "CfgCpp"   => [Cmem_dut,  [  ['Own',  ['Src',  [CMEMD,
DUT,],  [None]  ],],
                        ],
                   ],
        },
        }
);
```

**%PortProbeMap =**

3 of A1

```
(
 'cpu'=> {
         "FBUS"   => ["FBUS_PROBE" , Fbus_hwif ],
        },
);
######
1; ### makes perl happy
```

**ANNEX**

**A2 - Topological Rules Linked to the System Blocks and Intermediate Blocks**

```
#! /usr/triton/bin/perl
############################################################
###################
#
#      Copyright (c) 2000 BULL - Worldwide Information
Systems
#                         All rights reserved
#
#  Module name   :  patent_sys_config_defs.pm
#  Author        :  Andrzej Wozniak
#
#  Description   : tables for driving configuration
#          generation aux part
#
#  Rev Date : $Date: 2001-03-06 19:11:20+01 $
#
############################################################
###################
#########
$GlobSysInst           = 'SysClock';
$GlobSysInstModule     = 'Clock';
$GlobSysInstModuleSrc = 'patent_sys_glob.v';

# Factoring of the expression common to all of the
connections to the Global Blocks
@std_glob_con = (\&getGlobSysInst, "E",          1 of A2
           [\&subst_infix,
            ["^(?:clk)\$",   "sys_CLK_2X"],
            ["^(?:reset)\$",   "sys_RESET"],
            ["^(?:powergood)\$",   "sys_POWER_GOOD"],
            ],);

#######
%HwfConnectivityMap =                               2 of A2
  (
   "bridge" => {
     "cpu"      => "Direct",
     "cmem"     => "Direct",
     "cio"      => "Direct",
     "fbus_p"   => "fbus_xchg",
     "fbus_xchg" => { FBUS => 'Direct',
              },
   },

   "bridge_core" => {
     "cpu"      => "fbus_xchg",
     "cmem"     => "Direct",
      "cio"      => "Direct",
      "fbus_p"   => "Direct",
     "fbus_xchg" => { FBUSA => 'Direct', ,
              FBUS  => 'fbus_xchg',
                 },
    },

    "fbus_p"   => {
      "fbus_xchg" => { FBUSA => 'Direct',
                 FBUS => 'fbus_xchg',
               },
  "cpu"       => "fbus_xchg",
      "fbus_p"     => 'Direct',
    },

    "cio"     => { "cio_p"   => 'Direct' },
    "cmem"    => { "cmem_p" => 'Direct' },
    "fbus_xchg" => { "fbus_xchg" => { FBUS => 'Direct' },
   "cpu"        => { "fbus_xchg" => { FBUS => 'Direct' },
             },
    );
#######
%HwifAlternateMap =
(
 );
###
%SysWrapMap =
(
 bridge         => [ BRIDGE_sys,        sys_bridge  ],
 bridge_core    => [ BRIDGE_sys,        sys_bridge  ],
);

#######
%SysConnectMap =                            3 of A2
   (
    ######
    "$GlobSysInst" =>
    ######
    { "MatchExpr" => "^($GlobSysInst)\$",
      "ExtrExpr"  => "^($GlobSysInst)\$",
      ####
      "SYS_CON"    =>
      {"ReqModule" => {"$GlobSysInstModule" =>
"$GlobSysInstModuleSrc",
                 },
       "Connect"   => { "Global" => [\&subst_infix, ["(.*)",
""], ],
                 },
      },
    }, #### $GlobSysInst
    ######
 "BRIDGE_sys" =>
  ######
  { "MatchExpr" => "^(?:BRIDGE_[0-3]_)(sys)\$",
    "ExtrExpr"  => "^BRIDGE_([0-3])_sys\$",
    ####
    "SYS_CON"   =>
    { "ReqModule" =>
      {"sys_bridge" => "sys_bridge.v",
      },

      "Connect" => { BRIDGE_a => [\&subst_infix,  ["(.*)",
""],],
             },
      "SysConn" => { BRIDGE_a => [\&getGlobSysInst, "E",
[\&subst_infix,  ["sys_(.*)", ""],],],
             },
      "SysWrap" => { BRIDGE_a => [\&subst_infix,
["syswrap_(.*)", ""],],
             },
      ####
      # Construction of the names of the System Blocks
dedicated to a Component and
      # verification of the connectivity of the interfaces
between the System Blocks and
      # their dedicated Components
      "getOwnDest"   => \&getSpecSysOwn,
      "genVlogInstParameter" =>
\&gen_sys_VlogInstParameter,
      ####
    },
```

# Variables referring to a common generic code for all of the designs

```
  }, #### BRIDGE_sys
 );
###
%SysSpecMap =
  (
 ######
 "FBUS_XCHG" =>
 ######
 { "MatchExpr"    => "^(?:.*_)(FBUS_XCHG)\$",
   "ExtrExpr"     => "^(?:.*)([0-3])_FBUS_XCHG\$",
   "BaseIidExpr" => "^.*(FBUS_XCHG)\$",
  ###
   "IND_CON" =>
    { "ReqModule" =>
      {"fbus_xchg" => "fbus_xchg.v",
      },

      "Port" => { "FBUS" => ["FBUS_xchg",  "fbus_xchg", 0],
```

4 of A2

```
             "FBUSA" => ["FBUS_xchg",  "fbus_xchg", 1],
             },
       "Connect" => { "FBUS"   => [\&subst_infix,
["(XX.*)",    ""],],
                "FBUSA"  => [\&subst_infix,  ["in(.*)",
"aa_con"], ["out(.*)", "bb_con"],],
              },
       "SelfConnect"  => { "FBUS"   => [\&subst_infix,
["(XX.*)",    ""],],
                      "FBUSA"  => [\&subst_infix,
["out(.*)", "aa_con"], ["in(.*)", "bb_con"],],
               },
      "genOwn"      => \&genOwnSysSpecMapGeneric,
      "getOwnDest"  => \&getSpecSysOwn,

      "CfgCpp"  => [Fbus_Xchg, [ ['Own', [fbus_xchg,
IND_CON, ], ['None'], ],
                      ],
               ],
       },
     },
    ######
   'FBUS_PROBE' =>
    ######
    { "MatchExpr" => "^(.*)(FBUS_PROBE)\$",
      "ExtrExpr"  => "^.*?([0-3]).*FBUS_PROBE\$",
      "BaseIidExpr" => "^(.*)_FBUS_PROBE\$",
      ####
      "PROBE"   =>
      { "ReqModule" => {"fbus_probe" => "fbus_probe.v"
                },
       "Connect"   => {"FBUS"  => [\&subst_infix, ["(.*)",
""], ],
             },
       "Port" => { "FBUS"  => ["FBUS_probe",  "fbus_probe",
0],
             },
       "SysConn" => { "FBUS_probe" => \@std_glob_con,
                },
       "genVlogInstParameter" =>
\&gen_sys_VlogInstParameter_Generic,
       },
     },
);
###
%indTypeCftpMap =
(
 "FBUS_xchg" => "FBUS_XCHG",
 "fbus_xchg" => "FBUS_XCHG",
 );
```

5 of A2

```
###
@sysGlobConnTab =                      Referenced by %SysPinConst
  (
   [ sys_RESET          ,   RESET    ],
   [ sys_POWER_GOOD     ,    POWER_GOOD ],
   [ sys_CLK            ,    CLK_33MHz    ],
   [ sys_CLK_2X         ,    CLK_66MHz    ],
  );
###
%SysPinConst =                                           6 of A2
    (
     "$GlobSysInst" =>  \@sysGlobConnTab,
     CPU_sys   => \@sysGlobConnTab,
     CMEM_sys   => \@sysGlobConnTab,
     CIO_sys    => \@sysGlobConnTab,
     BRIDGE_sys    => \@sysGlobConnTab,
     BRIDGE => \@sysGlobConnTab,
     );

#####
sub gen_sys_VlogInstParameter {                           7 of A2
  my($inst, $cat, $Cftp, $i_map) = @_;
  ($Cftp, $i_map) = getCftp($inst) unless $Cftp and $i_map;
  my($extr) = $ {$i_map}{$Cftp}{'ExtrExpr'};
  $inst =~ $extr;
  my($Module_num, $Node_num, $SubNode_num) = ($1, $2, $3,
$4);
  my($parent_inst, $parent_cftp, $parent_map) =
&get_parent_inst($inst);
  my($num_add) = 0;
  $num_add = $ {$parent_map}{$parent_cftp}{'NumAdd'},
  $Node_num += $num_add
    if exists $ {$parent_map}{$parent_cftp}{'NumAdd'};
  my($parameter) = "";
  foreach $pp ($Module_num, $Node_num, $SubNode_num) {
    $parameter .= ", " if $parameter ne "" and $pp =~ /\d+/;
    $parameter .= $pp if $pp =~ /\d+/;
  }
  my($conn_mask) = &get_parent_inst_connected_mask($inst);
  if($conn_mask ne ""){
      $parameter .= ", " if $parameter ne "";
      $parameter .= $conn_mask;
  }
  if($parameter){
    return "#($parameter)";
  }
  return ""
}
#######
1; ### makes perl happy
```

**ANNEX**

**A3 - Procedures for Connecting th  Components**

```
#! /usr/triton/bin/perl
############################################################
###################
#
#      Copyright (c) 2000 BULL - Worldwide Information
Systems
#                            All rights reserved
#
#   Module name   :  patent_connect_defs.pm
#   Author        :  Andrzej Wozniak
#
#   Description   : connection generation
#
#   Revision : $Revision: 1.1 $
#
#   Rev Date : $Date: 2001-01-29 20:19:10+01 $
#
############################################################
###################

####
```

1 of A3

```
sub genDest {
  my($inst, $extExpr, $srcIfc) = @_;
  my(%dest) = ();
  my($dft);
  unless ( $inst =~ /$extExpr/ ){
   &dprint_config_error("genDest",
              "number extracting expression \"$extExpr\"
failed for $inst\n");
   return (\%dest, $dft);
  }
  while( $inst =~ /$extExpr/ ){
    $n1 = $1;
    $n2 = $2;
    $n3 = $3;
    my($nb) = 0;
    if($inst =~ $InstNameModuleMap{CPU}{"MatchExpr"}){
    ## CPU
      %dest = ("BRIDGE_$n1" => "FBUS"),
      $dft = "BRIDGE",
      last if $srcIfc =~ /^FBUS[A-Za-z0-9_]*/ ;
      goto label_error;
    }
    if($inst =~ $InstNameModuleMap{BRIDGE}{"MatchExpr"}){
    ## BRIDGE
      %dest = ( "CPU_$n1" => "FBUS"),
      $dft = "CPU",
      last if $srcIfc =~ /FBUS/;
```

```
        %dest = ( "CMEM_$n1"  => "CMEM"),
        $dft = "CMEM",
        last if $srcIfc =~ /CMEM/;
        %dest = ( "CIO_$n1"  => "CIO"),
        $dft = "CIO",
        last if $srcIfc =~ /CIO/;
        goto label_error;
      }
      if($inst =~ $InstNameModuleMap{CMEM}{"MatchExpr"}){
        ## CMEM
        %dest = ("BRIDGE_$n1" => "CMEM"),
        $dft = "BRIDGE",
        last if $srcIfc =~ /^CMEM$/;
        goto label_error;
      }
      if($inst =~ $InstNameModuleMap{CIO}{"MatchExpr"}){
        ## CIO
        %dest = ("BRIDGE_$n1" => "CIO"),
        $dft = "BRIDGE",
        last if $srcIfc =~ /^CIO$/;
        goto label_error;
      }
      &dprint_config_error("genDest", "unknown instance
$inst");
      last;
    label_error:
     &dprint_config_error("genDest", "unknown interface
$srcIfc for instance $inst");
      last;
    }
return (\%dest, $dft);
}
#######
                   1; ### makes perl happy
```

**ANNEX**

**A4 - Code for Generating the Component Connection File V rilog_Top.v**

```
#! /usr/triton/bin/perl
##############################################################
###################
#
#      Copyright (c) 2000 BULL - Worldwide Information
Systems
#                            All rights reserved
#
#  Module name  :  patent_verilog_defs.pm
#  Author       :  Andrzej Wozniak
#
#  Description  : VERILOG strings and defs
#                  for generation of top.v source code
#
#  Revision : $Revision: 1.4 $
#
#  Rev Date : $Date: 2001-02-19 19:55:41+01 $
#
##############################################################
###################

@verilog_src_path =
      (
       "$db_dir/patent/sim/models",
      );

$top_verilog_header = <<"END_OF_TOP_HEADER"
//////////////////////////////////////////////////////////////
//////////
//  FILE \"%s\" GENERATED by A.W. PERL SCRIPT
//  FROM \"%s\"  file
//////////////////////////////////////////////////////////////
//////////\n\n
//////
`timescale 100ps
END_OF_TOP_HEADER
;
;
$top_verilog_preamble = <<"END_OF_TOP_PREAMBLE"
//////
module top ();

wire       POWER_GOOD;
wire       RESET;

wire       CLK_33MHz;
wire        CLK_66MHz;

$GlobSysInstModule  $GlobSysInst(
        .sys_POWER_GOOD     (POWER_GOOD)
        .sys_RESET          (RESET),
        .sys_CLK            (CLK_33MHz),
        .sys_CLK_2X         (CLK_66MHz)
                    );
END_OF_TOP_PREAMBLE
;
####
```

**$top_verilog_postamble = <<"END_OF_TOP_POSTAMBLE"**

```
endmodule

/////////////
// END
/////////////
END_OF_TOP_POSTAMBLE
;

#######
1; ### makes perl happy
```

**ANNEX**

**A5 - Code for Generating $C^{++}$ Class s**

```
#! /usr/triton/bin/perl
############################################################
##################
#
#      Copyright (c) 2000 BULL - Worldwide Information
Systems
#                        All rights reserved
#
#  Module name   : patent_cpp_gen_defs.pm
#  Author        :  Andrzej Wozniak
#
#  Description   : cpp generation tables
#
#  Revision : $Revision: 1.1 $
#
############################################################
##################

####
%moduleToCpPRClassMap =
  (
   fbus_probe => Probe_hwif,
  );
####
%classCppProtoMap =
  (
   ####
   CPU_Dut =>
   ####
   {
     Prea => "static CPU_Dut #SrcInst#
(LabelClass::#SrcIid#, TypeClass::#SrcIct#,\n"
             ."\t\t\t\tstring(\"#SrcVlogPath#\"));\n",
   },
   ####
   CPU_Monitor =>
   ####
   {
     Prea => "static CPU_Monitor #SrcInst#
(LabelClass::#SrcIid#",
     Iter => ",\n\t\t\t&#DstPort#",
     Post => ");\n",
     Idle => ",\n\t\t\t0",
   },
   ####
   Bridge_Dut =>
   ####
   {
```

1 of A5

2 of A5

```
        Prea => "static Bridge_Dut #SrcInst#
(LabelClass::#SrcIid#, TypeClass::#SrcIct#,\n"
                ."\t\t\t\tstring(\"#SrcVlogPath#\"));\n",
    },
    ####
    Cio_Dut =>
    ####
    {
        Prea => "static Cio_Dut #SrcInst#
(LabelClass::#SrcIid#, TypeClass::#SrcIct#,\n"
                ."\t\t\t\tstring(\"#SrcVlogPath#\"));\n",
    },
    ####
    Cmem_Dut =>
    ####
    {
        Prea => "static Cmem_Dut #SrcInst#
(LabelClass::#SrcIid#, TypeClass::#SrcIct#,\n"
                ."\t\t\t\tstring(\"#SrcVlogPath#\"));\n",
    },
    ####
    CPU_Xactor =>
    ####
    {
        Prea => "static CPU_Xactor #SrcInst#
(LabelClass::#SrcIid#",
        Iter => ",\n\t\t\t&#SrcPort#",
        Idle => ",\n\t\t\t0",
        Post => ");\n",
    },
    ####
    Bridge_Xactor =>
    ####
    {
        Prea => "static Bridge_Xactor #SrcInst#
(LabelClass::#SrcIid#",
        Iter => ",\n\t\t\t&#SrcPort#",
        Idle => ",\n\t\t\t0",
        Post => ");\n",
    },
    ####
    Fbus_hwif =>
    ####
    {
Prea => "static Fbus_hwif #SrcInst# (LabelClass::#SrcIid#,
TypeClass::#SrcIct#,\n"
                ."\t\t\t\tstring(\"#SrcVlogPath#\"));\n",
    },
    ####
    Probe_hwif =>
    ####
```

```
   {
     Prea => "static Probe_hwif #SrcInst#
(LabelClass::#SrcIid#, TypeClass::PROBE,\n"
                .  "\t\t\t\tstring(\"#SrcVlogPath#\"));\n",
   },
   ####
   Cmem_hwif =>
   ####
   {
     Prea => "static Cmem_hwif #SrcInst#
(LabelClass::#SrcIid#, TypeClass::#SrcIct#,\n"
                .  "\t\t\t\tstring(\"#SrcVlogPath#\"));\n",
   },
   ####
   Cio_hwif =>
   ####
   {
     Prea => "static Cio_hwif #SrcInst#
(LabelClass::#SrcIid#, TypeClass::#SrcIct#,\n"
                .  "\t\t\t\tstring(\"#SrcVlogPath#\"));\n",
   },
   ####
   Fbus_Xchg =>
   ####
   {
     Prea => "static Fbus_Xchg #SrcInst#
(LabelClass::#SrcIid#, TypeClass::#SrcIct#,\n"
                .  "\t\t\t\tstring(\"#SrcVlogPath#\"));\n",
   },
   );
#####
$cpp_preamble = <<"END_OF_PREAMBLE"
////////////////////////////////////////
//  FILE GENERATED by A.W. PERL SCRIPT
//  FROM %s file
//  FOR %s
////////////////////////////////////////\n\n
/////////////
#include \"server_registry.hpp\"
#include \"server_components.hpp\"


/////////////\n\n
const string ServerRegistry::m_serverName = \"%s\";
const string ServerRegistry::m_configName = \"%s\";
const int ServerRegistry::m_serverNumber = %d;
Status InstantiateConfiguration()\{
//////////////\n
END_OF_PREAMBLE
;
####
```

3 of A5

4 of A5

```
$cpp_postamble = <<"END_OF_POSTAMBLE"
\treturn Success;\n}
//////////////
// END
//////////////
END_OF_POSTAMBLE
;
####

#######
1; ### makes perl happy
```

ANNEX

**A6 - Verilog Result File**

```
////////////////////////////////////////////////////////////
//////////
//  FILE "config_server_pat03_top.v" GENERATED by A.W. PERL
SCRIPT
//  FROM "patent/sim/configs/pat03.cfg"  file
////////////////////////////////////////////////////////////
//////////

//////
`timescale 100ps
//////
module top ();

wire        POWER_GOOD;
wire        RESET;

wire        CLK_33MHz;
wire        CLK_66MHz;

Clock  SysClock(
        .sys_POWER_GOOD    (POWER_GOOD)
        .sys_RESET         (RESET),
        .sys_CLK           (CLK_33MHz),
        .sys_CLK_2X        (CLK_66MHz)
                    );
     /////////////////////////////////
     ///// Wire Declaration Section
     /////////////////////////////////
// wire       CLK_33MHz;          // output(1)
// wire       CLK_66MHz;          // input(3) output(1)
// wire       POWER_GOOD;         // input(3) output(1)
// wire       RESET;         // input(3) output(1)
 wire     [3:0]     W1_00_inXXack;          // input(1)
output(1)
 wire     [63:0]    W1_00_inXXadr_dat;      // input(1)
output(1)
 wire     [3:0]     W1_00_inXXreq;          // input(1)
output(1)
 wire     [3:0]     W1_00_outXXack;         // input(1)
output(1)
 wire     [63:0]    W1_00_outXXadr_dat;          // input(1)
output(1)
 wire     [3:0]     W1_00_outXXreq;         // input(1)
output(1)
 wire     [3:0]     W_00_XXack;        // inout(2)
 wire     [63:0]    W_00_XXadr_dat;         // inout(2)
 wire     [3:0]     W_00_XXreq;        // inout(2)
 wire     [31:0]    W_00_YYadr;          // input(1)
output(1)
 wire     [2:0]     W_00_YYctrl;         // input(1)
output(1)
 wire     [63:0]    W_00_YYdata;         // inout(2)
 wire     [1:0]     W_00_ZZack;          // input(1)
output(1)
 wire     [15:0]    W_00_ZZdata;         // inout(2)
 wire     [1:0]     W_00_ZZreq;          // input(1)
output(1)
     /////////////////////////////////
 wire          W_00_clk_2xn;        // input(1) output(1)
 wire          W_00_clk_2xp;        // input(1) output(1)
 wire          W_00_clkn;           // input(1) output(1)
 wire          W_00_clkp;           // input(1) output(1)
 wire     [3:0]     W_00_inXXack;        // input(1)
output(1)
 wire     [63:0]    W_00_inXXadr_dat;         // input(1)
output(1)
 wire     [3:0]     W_00_inXXreq;        // input(1)
output(1)
 wire     [3:0]     W_00_outXXack;            // input(1)
output(1)
 wire     [63:0]    W_00_outXXadr_dat;        // input(1)
output(1)
 wire     [3:0]     W_00_outXXreq;            // input(1)
output(1)
 wire          W_00_powergood;           // input(1)
output(1)
 wire          W_00_reset;          // input(1) output(1)
     /////////////////////////////////
     ///// Module Instances Section
     /////////////////////////////////

//// BRIDGE_0_CPU_0_FBUS_XCHG -> IND_CON -> FBUS_xchg
////////////
fbus_xchg     BRIDGE_0_CPU_0_FBUS_XCHG (
           .XXadr_dat          (W_00_XXadr_dat),
           .XXreq         (W_00_XXreq),
           .XXack         (W_00_XXack),
           .inXXadr_dat        (W1_00_outXXadr_dat),
           .outXXadr_dat       (W1_00_inXXadr_dat),
           .inXXreq       (W1_00_outXXreq),
           .outXXreq           (W1_00_inXXreq),
           .inXXack       (W1_00_outXXack),
           .outXXack           (W1_00_inXXack));

//// CMEM_0 -> DUT -> CMEMD ////////////
cmem      CMEM_0 (
           .YYadr         (W_00_YYadr),
           .YYdata        (W_00_YYdata),
            .YYctrl        (W_00_YYctrl),
            .clk           (CLK_66MHz),
            .reset         (RESET),
            .powergood          (POWER_GOOD));

//// CIO_0 -> DUT -> CIOD ////////////
cio         CIO_0 (
            .ZZdata        (W_00_ZZdata),
            .ZZreq         (W_00_ZZreq),
            .ZZack         (W_00_ZZack),
            .clk           (CLK_66MHz),
            .reset         (RESET),
            .powergood          (POWER_GOOD));

//// BRIDGE_0 -> DUT_CORE -> BRD ////////////
bridge_core           BRIDGE_0 (
            .inXXadr_dat        (W1_00_inXXadr_dat),
            .outXXadr_dat       (W1_00_outXXadr_dat),
            .inXXreq       (W1_00_inXXreq),
            .outXXreq           (W1_00_outXXreq),
            .inXXack       (W1_00_inXXack),
            .outXXack           (W1_00_outXXack),
            .YYadr         (W_00_YYadr),
            .YYdata        (W_00_YYdata),
            .YYctrl        (W_00_YYctrl),
            .ZZdata        (W_00_ZZdata),
            .ZZreq         (W_00_ZZreq),
            .ZZack         (W_00_ZZack),
            .clk_2xp       (W_00_clk_2xp),
            .clk_2xn       (W_00_clk_2xn),
            .clkp          (W_00_clkp),
            .clkn          (W_00_clkn),
            .reset         (W_00_reset),
            .powergood          (W_00_powergood));

//// CPU_0_BRIDGE_0_FBUS_XCHG -> IND_CON -> FBUS_xchg
////////////
fbus_xchg        CPU_0_BRIDGE_0_FBUS_XCHG (
            .XXadr_dat          (W_00_XXadr_dat),
            .XXreq         (W_00_XXreq),
            .XXack         (W_00_XXack),
            .inXXadr_dat        (W_00_outXXadr_dat),
            .outXXadr_dat       (W_00_inXXadr_dat),
            .inXXreq       (W_00_outXXreq),
            .outXXreq           (W_00_inXXreq),
            .inXXack       (W_00_outXXack),
            .outXXack           (W_00_inXXack));

//// CPU_0 -> XACTOR -> FBUS_p ////////////
fbus_p           CPU_0_XACTOR_FBUS_p (
            .inXXadr_dat        (W_00_inXXadr_dat),
            .outXXadr_dat      (W_00_outXXadr_dat),
            .inXXreq       (W_00_inXXreq),
            .outXXreq          (W_00_outXXreq),
            .inXXack       (W_00_inXXack),
            .outXXack          (W_00_outXXack),
            .clk           (CLK_66MHz),
            .reset         (RESET),
            .powergood          (POWER_GOOD));

//// BRIDGE_0_sys -> SYS_CON -> BRIDGE_sys ////////////
sys_bridge     #(0, 32'h00000007)  BRIDGE_0_sys (
            .clk_2xp       (W_00_clk_2xp),
            .clk_2xn       (W_00_clk_2xn),
            .clkp          (W_00_clkp),
            .clkn          (W_00_clkn),
            .reset         (W_00_reset),
            .powergood          (W_00_powergood),
            .sys_CLK_2X         (CLK_66MHz),
            .sys_CLK       (CLK_33MHz),
            .sys_RESET          (RESET),
            .sys_POWER_GOOD          (POWER_GOOD));

endmodule

/////////////
// END
/////////////
```

ANNEX

**A7 - C++ Result Files**

```
////////////////////////////////////////
//  FILE GENERATED by A.W. PERL SCRIPT
//  FROM patent/sim/configs/pat03.cfg file
//  FOR server
////////////////////////////////////////

/////////////
#include "server_registry.hpp"
#include "server_components.hpp"


/////////////


const string ServerRegistry::m_serverName = "SERVER";
const string ServerRegistry::m_configName = "pat03";
const int ServerRegistry::m_serverNumber = 1;
Status InstantiateConfiguration(){
//////////////

static Fbus_hwif CPU_0_XACTOR_FBUS_p (LabelClass::CPU_0,
TypeClass::FBUS_type,
                    string("top.CPU_0_XACTOR_FBUS_p"));
static Cio_Dut CIO_0 (LabelClass::CIO_0, TypeClass::DUT,
                    string("top.CIO_0"));
static Cmem_Dut CMEM_0 (LabelClass::CMEM_0, TypeClass::DUT,
                    string("top.CMEM_0"));
static Bridge_Dut BRIDGE_0 (LabelClass::BRIDGE_0,
TypeClass::DUT_CORE,
                    string("top.BRIDGE_0"));
static CPU_Xactor CPU_0_XACTOR (LabelClass::CPU_0,
TypeClass::XACTOR,
                    &CPU_0_XACTOR_FBUS_p);
static CPU_Monitor CPU_0_MONITOR (LabelClass::CPU_0,
TypeClass::MONITOR,
                    &CPU_0_XACTOR_FBUS_p);
static Fbus_Xchg BRIDGE_0_CPU_0_FBUS_XCHG
(LabelClass::BRIDGE_0_CPU_0, TypeClass::FBUS_XCHG,
                    string("top.BRIDGE_0_CPU_0_FBUS_XCHG"));
static Fbus_Xchg CPU_0_BRIDGE_0_FBUS_XCHG
(LabelClass::CPU_0_BRIDGE_0, TypeClass::FBUS_XCHG,
                    string("top.CPU_0_BRIDGE_0_FBUS_XCHG"));
     return Success;
}
/////////////
// END
                    /////////////
```